Dinitz

```
|    struct Edge {
|        int x, y, c, f;
|    };
|
|    struct Flow {
|        vector<vector<int>> gf;
|        vector<Edge> edges;
|
|        explicit Flow(int n) {
|            gf.assign(n, {});
|        }
|
|        void add_edge(int x, int y, int c, bool directed) {
|            gf[x].push_back((int)edges.size());
|            edges.emplace_back(x, y, c, 0);
|            gf[y].push_back((int)edges.size());
|            edges.emplace_back(y, x, (1-directed)*c, 0);
|        }
|
|        vector<int> layer, ind;
|
|        bool build_layers(int x, int y) {
|            layer.assign(gf.size(), -1);
|            ind.assign(gf.size(), 0);
|            queue<int> a;
|            a.push(x);
|            layer[x] = 0;
|            while (!a.empty()) {
|                int q = a.front();
|                if (q == y) {
|                    return true;
|                }
|                a.pop();
|                for (int q1_ : gf[q]) {
|                    Edge& q1 = edges[q1_];
|                    if (q1.c == q1.f) {
|                        continue;
|                    }
|                    if (layer[q1.y] == -1) {
|                        a.push(q1.y);
|                        layer[q1.y] = layer[q1.x]+1;
|                    }
|                }
|            }
|            return layer[y] != -1;
|        }
|
|        int push(int x, int y, int min1) {
|            if (x == y) {
|                return min1;
|            }
|            int ans = 0;
|            for (; ind[x] < gf[x].size(); ind[x]++) {
|                int num = gf[x][ind[x]];
|                Edge& q = edges[num];
|                if (layer[q.y] != layer[q.x]+1 || q.f == q.c) {
|                    continue;
|                }
|                int pushed = push(q.y, y, min(min1, q.c-q.f));
|                edges[num].f += pushed;
|                edges[num ^ 1].f -= pushed;
|                ans += pushed, min1 -= pushed;
```

```
                if (min1 == 0) {
                    return ans;
                }
            }
            return ans;
        }

        void build_flow(int x, int y) {
            while (build_layers(x, y)) {
                push(x, y, INF);
            }
        }

        int max_flow(int x, int y) {
            build_flow(x, y);
            int ans = 0;
            for (int q : gf[x]) {
                ans += edges[q].f;
            }
            return ans;
        }
    };
```

Ford_Fulkerson

```
    struct Edge {
        int x, y, c, f = 0;
    };

    vector<vector<Edge>> d;
    vector<vector<int>> gf;
    vector<Edge> e;
    vector<bool> was;

    int push(int vertex, int x, int y) {
        if (vertex == y) {
            return x;
        }
        was[vertex] = true;
        for (int q : gf[vertex]) {
            Edge &q1 = e[q];
            if (!was[q1.y] && q1.c-q1.f > 0) {
                int x1 = push(q1.y, min(x, q1.c-q1.f), y);
                if (x1 != -1) {
                    q1.f += x1, e[q ^ 1].f -= x1;
                    return x1;
                }
            }
        }
        return -1;
    }

    bool cancel_way(int vertex, int y) {
        if (vertex == y) {
            return true;
        }
        was[vertex] = true;
        for (int q : gf[vertex]) {
            Edge &q1 = e[q];
            if (!was[q1.y] && q1.f > 0 && cancel_way(q1.y, y)) {
                q1.f--, e[q ^ 1].f++;
                return true;
            }
        }
```

```
|        return false;
|    }
|
|    void cancel(int ind, int x, int y) {
|        if (e[ind].f == 0) {
|            return;
|        }
|        e[ind].c--, e[ind].f--;
|        was.assign(d.size(), false);
|        assert(cancel_way(x, e[ind].x));
|        was.assign(d.size(), false);
|        assert(cancel_way(e[ind].y, y));
|    }
|
|    void build_flow() {
|        int n = (int)d.size();
|        gf.assign(n, {}), e = {};
|        for (int q = 0; q < n; q++) {
|            for (Edge q1 : d[q]) {
|                gf[q1.x].push_back((int)e.size());
|                e.emplace_back(q1.x, q1.y, q1.c);
|                gf[q1.y].push_back((int)e.size());
|                e.emplace_back(q1.y, q1.x, 0);
|            }
|        }
|    }
|
|    int max_flow(int x, int y) {
|        int n = (int)d.size();
|        int size = 0;
|        while (size != -1) {
|            was.assign(n, false);
|            size = push(x, INF, y);
|        }
|        int ans = 0;
|        for (int q : gf[x]) {
|            ans += e[q].f;
|        }
|        return ans;
|    }
```

Min_cost

```
|    struct Edge {
|        int x, y, c, f, v;
|    };
|
|    struct Flow {
|        vector<vector<int>> gf;
|        vector<Edge> edges;
|
|        explicit Flow(int n) {
|            gf.assign(n, {});
|        }
|
|        void add_edge(int x, int y, int c, int v) {
|            gf[x].push_back((int)edges.size());
|            edges.emplace_back(x, y, c, 0, v);
|            gf[y].push_back((int)edges.size());
|            edges.emplace_back(y, x, 0, 0, -v);
|        }
|
|        vector<int> dists;
|
```

```
|        void build_dists(int x) {
|            int n = (int)gf.size();
|            dists.assign(n, INF);
|            vector<bool> taken(n, false);
|            queue<int> a;
|            dists[x] = 0, taken[x] = true;
|            a.push(x);
|            while (!a.empty()) {
|                int q = a.front();
|                a.pop();
|                taken[q] = false;
|                for (int q1 : gf[q]) {
|                    Edge& e = edges[q1];
|                    if (e.f != e.c && dists[e.y] > dists[q]+e.v) {
|                        dists[e.y] = dists[q]+e.v;
|                        if (!taken[e.y]) {
|                            taken[e.y] = true;
|                            a.push(e.y);
|                        }
|                    }
|                }
|            }
|        }
|
|        bool push(int x, int y) {
|            int n = (int)gf.size();
|            vector<int> will(n, INF), parents(n, -1);
|            priority_queue<p> a;
|            will[x] = 0;
|            a.emplace(0, x);
|            while (!a.empty()) {
|                int len = -a.top().first, q = a.top().second;
|                a.pop();
|                if (len != will[q]) {
|                    continue;
|                }
|                for (int q1 : gf[q]) {
|                    Edge& e = edges[q1];
|                    int will_dist = len+e.v+dists[e.x]-dists[e.y];
|                    if (e.f != e.c && will[e.y] > will_dist) {
|                        will[e.y] = will_dist, parents[e.y] = q1;
|                        a.emplace(-will_dist, e.y);
|                    }
|                }
|            }
|            if (will[y] == INF) {
|                return false;
|            }
|            while (x != y) {
|                edges[parents[y]].f++;
|                edges[parents[y] ^ 1].f--;
|                y = edges[parents[y]].x;
|            }
|            for (int q = 0; q < n; q++) {
|                will[q] -= dists[x]-dists[q];
|            }
|            dists = will;
|            return true;
|        }
|
|        void build_flow(int x, int y, int k) {
|            build_dists(x);
|            for (int q = 0; q < k && push(x, y); q++);
|        }
```

```
|
|       int min_cost(int x, int y, int k = INF) {
|           build_flow(x, y, k);
|           int ans = 0;
|           for (int q = 0; q < edges.size(); q += 2) {
|               ans += edges[q].f*edges[q].v;
|           }
|           return ans;
|       }
|
|       vector<vector<int>> ways;
|
|       void find_way(int x, int y) {
|           if (x == y) {
|               return;
|           }
|           for (int q1 : gf[x]) {
|               Edge& q = edges[q1];
|               if (q.f > 0) {
|                   find_way(q.y, y);
|                   edges[q1].f--, edges[q1 ^ 1].f++;
|                   ways.back().push_back(q1);
|                   return;
|               }
|           }
|       }
|
|       void decompose(int x, int y) {
|           ways = {};
|           vector<Edge> was_edges = edges;
|           int k = 0;
|           for (int q : gf[x]) {
|               k += edges[q].f;
|           }
|           for (int q = 0; q < k; q++) {
|               ways.emplace_back();
|               find_way(x, y);
|               reverse(ways.back().begin(), ways.back().end());
|           }
|           edges = was_edges;
|       }
|   };
```

LCA_linear_memory

```
|   vector<vector<int>> d;
|   vector<int> parent, height, jump;
|
|   void make_LCA(int vertex, int p1, int h) {
|       parent[vertex] = p1, height[vertex] = h;
|       if (p1 != -1 && height[jump[p1]]-height[p1] == height[jump[jump[p1]]]-height[jump[p1]]) {
|           jump[vertex] = jump[jump[p1]];
|       } else {
|           jump[vertex] = (p1 == -1 ? vertex : p1);
|       }
|       for (int q : d[vertex]) {
|           if (q != p1) {
|               make_LCA(q, vertex, h+1);
|           }
|       }
|   }
|
|   int k_ancestor(int vertex, int k) {
|       int h = height[vertex]-k;
```

```
|        while (height[vertex] > h) {
|            vertex = (height[jump[vertex]] >= h ? jump[vertex] : parent[vertex]);
|        }
|        return vertex;
|    }
|
|    int LCA(int x, int y) {
|        if (height[x] < height[y]) {
|            swap(x, y);
|        }
|        x = k_ancestor(x, height[x]-height[y]);
|        while (x != y) {
|            if (jump[x] != jump[y]) {
|                x = jump[x], y = jump[y];
|            } else {
|                x = parent[x], y = parent[y];
|            }
|        }
|        return x;
|    }
```

Compressed_tree

```
|    vector<vector<int>> tree;
|
|    int make_compressed_tree(vector<int> a) {
|        int n = a.size();
|        sort(a.begin(), a.end(), [](int x, int y) {return tin[x] < tin[y];});
|        for (int q = 1; q < n; q++) {
|            a.push_back(LCA(a[q-1], a[q]));
|        }
|        sort(a.begin(), a.end(), [](int x, int y) {return tin[x] < tin[y];});
|        for (int q : a) {
|            tree[q] = {};
|        }
|        n = a.size();
|        vector<int> stack = {a[0]};
|        for (int q = 1; q < n; q++) {
|            while (tout[stack.back()] < tin[a[q]]) {
|                stack.pop_back();
|            }
|            if (stack.back() != a[q]) {
|                tree[stack.back()].push_back(a[q]);
|                tree[a[q]].push_back(stack.back());
|                stack.push_back(a[q]);
|            }
|        }
|        return a[0];
|    }
```

HLD

```
|    vector<vector<int>> d;
|    vector<int> parent, sizes, tin, tout, height, order, up;
|    vector<int> a;
|
|    void for_HLD(int vertex, int p1) {
|        int now = 1;
|        for (int q : d[vertex]) {
|            if (q != p1) {
|                for_HLD(q, vertex);
|                now += sizes[q];
|            }
```

```
        }
        sizes[vertex] = now, parent[vertex] = p1;
        sort(d[vertex].begin(), d[vertex].end(), [](int x, int y) {return sizes[x] > sizes[y];});
    }

    void DFS_HLD(int vertex, int p1, int h) {
        tin[vertex] = order.size(), height[vertex] = h;
        up[vertex] = (p1 == -1 || d[p1][0] != vertex ? vertex : up[p1]);
        order.push_back(a[vertex]);
        for (int q : d[vertex]) {
            if (q != p1) {
                DFS_HLD(q, vertex, h+1);
            }
        }
        tout[vertex] = order.size();
    }

    DO make_HLD() {
        int n = d.size();
        sizes.assign(n, -1), parent.assign(n, -1), height.assign(n, -1);
        tin.assign(n, -1), tout.assign(n, -1), up.assign(n, -1);
        for_HLD(0, -1);
        DFS_HLD(0, -1, 0);
        return DO(order);
    }

    void update_way(int x, int y, int k, DO &do_arr) {
        if (height[up[x]] < height[up[y]]) {
            swap(x, y);
        }
        while (up[x] != up[y]) {
            do_arr.update(0, do_arr.len, tin[up[x]], tin[x]+1, 1, k);
            x = parent[up[x]];
            if (height[up[x]] < height[up[y]]) {
                swap(x, y);
            }
        }
        do_arr.update(0, do_arr.len, min(tin[x], tin[y]), max(tin[x], tin[y])+1, 1, k);
    }

    int ans_way(int x, int y, DO &do_arr) {
        if (height[up[x]] < height[up[y]]) {
            swap(x, y);
        }
        int ans = 0;
        while (up[x] != up[y]) {
            ans += do_arr.ans(0, do_arr.len, tin[up[x]], tin[x]+1, 1);
            x = parent[up[x]];
            if (height[up[x]] < height[up[y]]) {
                swap(x, y);
            }
        }
        ans += do_arr.ans(0, do_arr.len, min(tin[x], tin[y]), max(tin[x], tin[y])+1, 1);
        return ans;
    }
```

Bridges

```
    struct Edge {
        int x, y, num;

        Edge(int x1, int y1, int num1): x(x1), y(y1), num(num1) {}
    };
```

```
|    bool operator<(Edge edge, Edge edge1) {
|        return edge.num < edge1.num;
|    }
|
|    vector<vector<Edge>> d;
|    vector<int> tin, up;
|    vector<bool> was;
|    int time1 = 0;
|    set<Edge> bridge;
|
|    void bridges(int vertex, int num) {
|        tin[vertex] = up[vertex] = time1++, was[vertex] = true;
|        for (Edge q : d[vertex]) {
|            if (!was[q.y]) {
|                bridges(q.y, q.num);
|                up[vertex] = min(up[vertex], up[q.y]);
|                if (up[q.y] > tin[vertex]) {
|                    bridge.insert(q);
|                }
|            } else if (q.num != num) {
|                up[vertex] = min(up[vertex], tin[q.y]);
|            }
|        }
|    }
|
|    vector<vector<pair<int, p>>> d1;
|    vector<vector<int>> comp;
|    vector<p> num;
|
|    void cond(int vertex, int now) {
|        was[vertex] = false, num[vertex] = {now, comp[now].size()};
|        comp[now].push_back(vertex);
|        for (Edge q: d[vertex]) {
|            if (!was[q.y] && bridge.find(q) != bridge.end()) {
|                cond(q.y, now);
|            } else if (!was[q.y]) {
|                d1.emplace_back();
|                d1[now].push_back({comp.size(), {q.x, q.y}});
|                d1[comp.size()].push_back({now, {q.y, q.x}});
|                comp.emplace_back();
|                cond(q.y, (int)comp.size()-1);
|            }
|        }
|    }
```

CSS

```
|    vector<vector<int>> d, d1, cond, comp;
|    vector<int> who, topsort;
|    vector<bool> was;
|
|    void top_sort(int vertex) {
|        was[vertex] = true;
|        for (int q : d[vertex]) {
|            if (!was[q]) {
|                top_sort(q);
|            }
|        }
|        topsort.push_back(vertex);
|    }
|
|    void DFS_CSS(int vertex) {
|        was[vertex] = true, who[vertex] = (int)comp.size()-1;
|        comp.back().push_back(vertex);
```

```
        for (int q : d1[vertex]) {
            if (!was[q]) {
                DFS_CSS(q);
            } else if (who[q] != who[vertex]) {
                cond[who[q]].push_back(who[vertex]);
            }
        }
    }

    void CSS() {
        int n = (int)d.size();
        d1.assign(n, {}), comp = {}, cond = {};
        who.assign(n, -1), topsort = {}, was.assign(n, false);
        for (int q = 0; q < n; q++) {
            if (!was[q]) {
                top_sort(q);
            }
            for (int q1 : d[q]) {
                d1[q1].push_back(q);
            }
        }
        reverse(topsort.begin(), topsort.end());
        was.assign(n, false);
        for (int q : topsort) {
            if (!was[q]) {
                comp.emplace_back();
                cond.emplace_back();
                DFS_CSS(q);
            }
        }
    }
```

Points_articulation

```
    vector<vector<int>> d, comp;
    vector<int> tin, up;
    vector<pair<int, bool>> bypass;
    int time1;

    void build_comp(int vertex) {
        comp.push_back({vertex});
        while (bypass.back().first != vertex) {
            if (bypass.back().second) {
                comp.back().push_back(bypass.back().first);
            }
            bypass.pop_back();
        }
    }

    void DFS_articulation(int vertex, int parent) {
        tin[vertex] = up[vertex] = time1++;
        bypass.emplace_back(vertex, true);
        for (int q : d[vertex]) {
            if (tin[q] == -1) {
                bypass.emplace_back(vertex, false);
                DFS_articulation(q, vertex);
                if (up[q] >= tin[vertex]) {
                    build_comp(vertex);
                }
                up[vertex] = min(up[vertex], up[q]);
            } else if (q != parent) {
                up[vertex] = min(up[vertex], tin[q]);
            }
        }
```

```
    }

    vector<vector<int>> g;

    int make_articulation(int n) {
        tin.assign(n, -1), up.assign(n, -1);
        comp = {}, bypass = {}, time1 = 0;
        DFS_articulation(0, -1);
        int k = (int)comp.size();
        g.assign(n+k, {});
        for (int q = 0; q < k; q++) {
            for (int q1 : comp[q]) {
                g[q+n].push_back(q1);
                g[q1].push_back(q+n);
            }
        }
        return k;
    }
```

Eulerian_cycle

```
    vector<int> Euler;
    vector<set<int>> copy_d;

    void make_bypass(int vertex) {
        while (!copy_d[vertex].empty()) {
            int q = *copy_d[vertex].begin();
            copy_d[vertex].erase(q);
            copy_d[q].erase(vertex);
            make_bypass(q);
        }
        Euler.push_back(vertex);
    }
```

Kun

```
    vector<vector<int>> d;
    vector<int> pa, pb;
    vector<bool> was_a, was_b;

    bool find_chain(int vertex) {
        was_a[vertex] = true;
        for (int q : d[vertex]) {
            if (pb[q] == -1 || !was_a[pb[q]] && find_chain(pb[q])) {
                pa[vertex] = q, pb[q] = vertex;
                return true;
            }
        }
        return false;
    }

    mt19937 randint(17957179);

    void Kun(int n, int m) {
        pa.assign(n, -1), pb.assign(n+m, -1);
        vector<int> perm(n);
        iota(perm.begin(), perm.end(), 0);
        shuffle(perm.begin(), perm.end(), randint);
        bool flag = true;
        while (flag) {
            flag = false;
            was_a.assign(n, false);
            for (int q : perm) {
```

```
|                 if (pa[q] == -1 && find_chain(q)) {
|                     flag = true;
|                 }
|             }
|         }
|     }
|
|     int max_matching(int n, int m) {
|         Kun(n, m);
|         return n-count(pa.begin(), pa.end(), -1);
|     }
|
|     void DFS_L_minus(int vertex) {
|         was_a[vertex] = true;
|         for (int q : d[vertex]) {
|             if (q == pa[vertex] || was_b[q]) {
|                 continue;
|             }
|             was_b[q] = true;
|             if (pb[q] != -1 && !was_a[pb[q]]) {
|                 DFS_L_minus(pb[q]);
|             }
|         }
|     }
|
|     int independent_set(int n, int m) {
|         Kun(n, m);
|         was_a.assign(n, false), was_b.assign(m, false);
|         for (int q = 0; q < n; q++) {
|             if (pa[q] == -1) {
|                 DFS_L_minus(q);
|             }
|         }
|         return count(was_a.begin(), was_a.end(), true)+count(was_b.begin(), was_b.end(), false);
|     }
|
|     int paths_splitting(int n) {
|         return n-max_matching(n, n);
|     }
|
|     void DFS_reachable(int vertex) {
|         was_a[vertex] = true;
|         for (int q : d[vertex]) {
|             if (!was_a[q]) {
|                 DFS_reachable(q);
|             }
|         }
|     }
|
|     void make_transitive_closure(int n) {
|         for (int q = 0; q < n; q++) {
|             was_a.assign(n, false);
|             DFS_reachable(q);
|             d[q] = {};
|             for (int q1 = 0; q1 < n; q1++) {
|                 if (q != q1 && was_a[q1]) {
|                     d[q].push_back(q1);
|                 }
|             }
|         }
|     }
|
|     int max_antichain(int n) {
|         make_transitive_closure(n);
```

```
|        return independent_set(n, n)-n; // was_a && !was_b
|    }
```

Pollard

```
|    __int128 pow2(__int128 x, __int128 y, __int128 C) {
|        if (y == 0) {
|            return 1;
|        }
|        if (y % 2 == 0) {
|            return pow2(x*x % C, y/2, C);
|        }
|        return pow2(x, y-1, C)*x % C;
|    }
|
|    bool Miller_Rabin_test(int a, int n) {
|        int d = n-1;
|        while ((d & 1) ^ 1) {
|            d >>= 1;
|        }
|        __int128 now = pow2(a, d, n);
|        if (now == 1) {
|            return true;
|        }
|        while (d < n-1) {
|            if (now == n-1) {
|                return true;
|            }
|            now = now*now % n, d <<= 1;
|        }
|        return false;
|    }
|
|    mt19937 randint(17957179);
|
|    bool Miller_Rabin(int n, int k = 20) {
|        if (n == 1) {
|            return false;
|        }
|        for (int q = 0; q < k; q++) {
|            if (!Miller_Rabin_test(randint() % (n-1)+1, n)) {
|                return false;
|            }
|        }
|        return true;
|    }
|
|    int f_Pollard(__int128 x, int n) {
|        return (x*x+3) % n;
|    }
|
|    vector<int> make_Pollard(int n) {
|        if (Miller_Rabin(n)) {
|            return {n};
|        }
|        int x = randint() % (n-1)+1;
|        int y = f_Pollard(x, n);
|        while (__gcd(n, abs(y-x)) == 1) {
|            x = f_Pollard(x, n);
|            y = f_Pollard(f_Pollard(y, n), n);
|        }
|        if (x == y) {
|            return make_Pollard(n);
|        }
```

```
|        int d = __gcd(n, abs(y-x));
|        vector<int> ans = make_Pollard(d);
|        for (int q : make_Pollard(n/d)) {
|            ans.push_back(q);
|        }
|        return ans;
|    }
|
|    vector<int> Pollard(int n) {
|        vector<int> primes, small = {2, 3, 5, 7};
|        for (int q : small) {
|            while (n % q == 0) {
|                primes.push_back(q);
|                n /= q;
|            }
|        }
|        if (n == 1) {
|            return primes;
|        }
|        multiset<int> all;
|        for (int q : make_Pollard(n)) {
|            all.insert(q);
|        }
|        for (int q : all) {
|            primes.push_back(q);
|        }
|        return primes;
|    }
```

FFT_divide

```
|    int pow1(int x, int y, int C) {
|        if (y == 0) {
|            return 1;
|        }
|        if (y % 2 == 0) {
|            return pow1(x*x % C, y/2, C);
|        }
|        return pow1(x, y-1, C)*x % C;
|    }
|
|    #define ld double
|    const ld PI = acosl(-1);
|
|    int reverse_bits(int x, int len) {
|        int y = 0;
|        for (int q = 0; q < len; q++) {
|            y |= (((x >> q) & 1) << (len-q-1));
|        }
|        return y;
|    }
|
|    void FFT(vector<complex<ld>> &a) {
|        int n = a.size(), len = 1;
|        while ((1 << len) < n) {
|            len++;
|        }
|        for (int q = 0; q < n; q++) {
|            int q1 = reverse_bits(q, len);
|            if (q < q1) {
|                swap(a[q], a[q1]);
|            }
|        }
|        for (int q = 1; q < n; q <<= 1) {
```

```
            complex<ld> root(cosl(PI/q), sinl(PI/q));
            for (int q1 = 0; q1 < n; q1 += (q << 1)) {
                complex<ld> now = 1;
                for (int q2 = q1; q2 < q1+q; q2++) {
                    complex<ld> x = a[q2], y = a[q2+q]*now;
                    a[q2] = x+y, a[q2+q] = x-y;
                    now *= root;
                }
            }
        }
    }

    void IFFT(vector<complex<ld>> &a) {
        int n = a.size();
        FFT(a);
        reverse(a.begin()+1, a.end());
        for (complex<ld> &q : a) {
            q /= n;
        }
    }

    vector<complex<ld>> to_FFT_form(const vector<int> &a, int deg) {
        vector<complex<ld>> A;
        for (int q : a) {
            A.push_back(q);
        }
        while (A.size() < deg) {
            A.push_back(0);
        }
        return A;
    }

    vector<int> from_FFT_form(const vector<complex<ld>> &A) {
        vector<int> a;
        for (const complex<ld> &q : A) {
            a.push_back(roundl(q.real()));
        }
        while (a.size() > 1 && a.back() == 0) {
            a.pop_back();
        }
        return a;
    }

    vector<int> multiply(const vector<int> &a, const vector<int> &b, int MOD) {
        int deg = 1;
        while (deg < a.size()+b.size()) {
            deg <<= 1;
        }
        vector<complex<ld>> A = to_FFT_form(a, deg);
        vector<complex<ld>> B = to_FFT_form(b, deg);
        FFT(A), FFT(B);
        vector<complex<ld>> C(deg);
        for (int q = 0; q < deg; q++) {
            C[q] = A[q]*B[q];
        }
        IFFT(C);
        vector<int> c = from_FFT_form(C);
        for (int &q : c) {
            q %= MOD;
        }
        return c;
    }

    void fix_zeros(vector<int> &a, int n) {
```

```
|        while (a.size() < n) {
|            a.push_back(0);
|        }
|        while (a.size() > n && a.back() == 0) {
|            a.pop_back();
|        }
|    }
|
|    void modulo_x_n(vector<int> &a, int n) {
|        while (a.size() < n) {
|            a.push_back(0);
|        }
|        while (a.size() > n) {
|            a.pop_back();
|        }
|    }
|
|    int get_degree(int n) {
|        int deg = 1;
|        while (deg < n) {
|            deg <<= 1;
|        }
|        return deg;
|    }
|
|    vector<int> opposite(vector<int> a, int MOD) {
|        for (int &q : a) {
|            q = (MOD-q) % MOD;
|        }
|        return a;
|    }
|
|    vector<int> find_reversed(vector<int> &a, int n, int MOD) {
|        if (n == 1) {
|            if (a[0] == 0) {
|                exit(179);
|            }
|            return {pow1(a[0], MOD-2, MOD)};
|        }
|        vector<int> a0, a1;
|        for (int q = 0; q < n/2; q++) {
|            a0.push_back(a[q]);
|        }
|        for (int q = n/2; q < n; q++) {
|            a1.push_back(a[q]);
|        }
|        vector<int> rev_a0 = find_reversed(a0, n/2, MOD);
|        vector<int> Q = multiply(a0, rev_a0, MOD);
|        fix_zeros(Q, n);
|        reverse(Q.begin(), Q.end());
|        modulo_x_n(Q, n/2);
|        reverse(Q.begin(), Q.end());
|        vector<int> rev_a0_by_a1 = multiply(rev_a0, a1, MOD);
|        modulo_x_n(rev_a0_by_a1, n/2);
|        for (int q = 0; q < n/2; q++) {
|            Q[q] += rev_a0_by_a1[q];
|        }
|        vector<int> E = multiply(rev_a0, Q, MOD);
|        modulo_x_n(E, n/2);
|        E = opposite(E, MOD);
|        for (int q : E) {
|            rev_a0.push_back(q);
|        }
|        return rev_a0;
```

```
|    }
|
|    vector<int> reverse(vector<int> a, int n, int MOD) {
|        int deg = get_degree(n);
|        fix_zeros(a, deg);
|        vector<int> rev = find_reversed(a, deg, MOD);
|        modulo_x_n(rev, n);
|        return rev;
|    }
|
|    vector<int> divide(vector<int> a, vector<int> b, int MOD) {
|        if (a.size() < b.size() || a.size() == 1 && a[0] == 0) {
|            return {0};
|        }
|        if (b.size() == 1 && b[0] == 0) {
|            exit(179);
|        }
|        reverse(a.begin(), a.end());
|        reverse(b.begin(), b.end());
|        vector<int> rev = reverse(b, (int)a.size()-(int)b.size()+1, MOD);
|        vector<int> c = multiply(a, rev, MOD);
|        modulo_x_n(c, (int)a.size()-(int)b.size()+1);
|        reverse(c.begin(), c.end());
|        return c;
|    }
|
|    vector<int> add(const vector<int> &a, const vector<int> &b, int MOD) {
|        int n = a.size(), m = b.size();
|        vector<int> c(max(n, m));
|        for (int q = 0; q < max(n, m); q++) {
|            c[q] = ((q < n ? a[q] : 0)+(q < m ? b[q] : 0)) % MOD;
|        }
|        fix_zeros(c, 1);
|        return c;
|    }
|
|    vector<int> subtract(const vector<int> &a, const vector<int> &b, int MOD) {
|        return add(a, opposite(b, MOD), MOD);
|    }
|
|    vector<int> take_remainder(const vector<int> &a, const vector<int> &b, int MOD) {
|        vector<int> divider = divide(a, b, MOD);
|        return subtract(a, multiply(b, divider, MOD), MOD);
|    }
|
|    vector<int> pow(vector<int> P, vector<int> mod, int n, int MOD) {
|        if (n == 0) {
|            return {1};
|        }
|        if (n % 2 == 0) {
|            return pow(take_remainder(multiply(P, P, MOD), mod, MOD), mod, n/2, MOD);
|        }
|        return take_remainder(multiply(pow(P, mod, n-1, MOD), P, MOD), mod, MOD);
|    }
```

FFT_modulo

```
|    const int g = 31;
|    vector<int> rev_bits;
|
|    int reverse_bits(int x, int log1) {
|        int y = 0;
|        for (int q = 0; q < log1; q++) {
|            y |= (((x >> q) & 1) << (log1-q-1));
```

```
|        }
|        return y;
|    }
|
|    void build_rev_bits(int log1) {
|        int n = (1 << log1);
|        rev_bits.assign(n, -1);
|        for (int q = 0; q < n; q++) {
|            int q1 = reverse_bits(q, log1);
|            rev_bits[q] = min(q, q1);
|        }
|    }
|
|    void FFT(int* a, int log1) {
|        int n = (1 << log1);
|        for (int q = 0; q < n; q++) {
|            swap(a[q], a[rev_bits[q]]);
|        }
|        for (int q = 1; q < n; q <<= 1) {
|            int root = pow1(g, (C-1)/(q << 1));
|            for (int q1 = 0; q1 < n; q1 += (q << 1)) {
|                int now = 1;
|                for (int q2 = q1; q2 < q1+q; q2++) {
|                    int x = a[q2], y = a[q2+q]*now % C;
|                    a[q2] = x+y-(x+y >= C)*C;
|                    a[q2+q] = x-y+(x-y < 0)*C;
|                    now = now*root % C;
|                }
|            }
|        }
|    }
|
|    void IFFT(int* a, int log1) {
|        int n = (1 << log1);
|        FFT(a, log1);
|        reverse(a+1, a+n);
|        int rev_n = pow1(n, C-2);
|        for (int q = 0; q < n; q++) {
|            a[q] *= rev_n, a[q] %= C;
|        }
|    }
|
|    int get_degree(int n) {
|        int log1 = 0;
|        while ((1 << log1) < n) {
|            log1++;
|        }
|        return log1;
|    }
|
|    vector<int> multiply(vector<int>& aa, vector<int>& bb) {
|        int len = (int)aa.size()+(int)bb.size()-1;
|        int log1 = get_degree(len), n = (1 << log1);
|        build_rev_bits(log1);
|        int a[n], b[n];
|        fill(a, a+n, 0), fill(b, b+n, 0);
|        copy(aa.begin(), aa.end(), a);
|        copy(bb.begin(), bb.end(), b);
|        FFT(a, log1), FFT(b, log1);
|        for (int q = 0; q < n; q++) {
|            a[q] = a[q]*b[q] % C;
|        }
|        IFFT(a, log1);
|        vector<int> ans(a, a+len);
```

```
|        return ans;
|    }
```

online_FFT

```
|    const int g = 31;
|
|    int reverse_bits(int x, int log1) {
|        int y = 0;
|        for (int q = 0; q < log1; q++) {
|            y |= (((x >> q) & 1) << (log1-q-1));
|        }
|        return y;
|    }
|
|    void FFT(vector<int>& a, int log1) {
|        int n = (1 << log1);
|        for (int q = 0; q < n; q++) {
|            int rev = reverse_bits(q, log1);
|            swap(a[q], a[min(q, rev)]);
|        }
|        for (int q = 1; q < n; q <<= 1) {
|            int root = pow1(g, (C-1)/(q << 1));
|            for (int q1 = 0; q1 < n; q1 += (q << 1)) {
|                int now = 1;
|                for (int q2 = q1; q2 < q1+q; q2++) {
|                    int x = a[q2], y = a[q2+q]*now % C;
|                    a[q2] = x+y-(x+y >= C)*C;
|                    a[q2+q] = x-y+(x-y < 0)*C;
|                    now = now*root % C;
|                }
|            }
|        }
|    }
|
|    void IFFT(vector<int>& a, int log1) {
|        int n = (1 << log1);
|        FFT(a, log1);
|        reverse(a.begin()+1, a.end());
|        int rev_n = pow1(n, C-2);
|        for (int& q : a) {
|            q *= rev_n, q %= C;
|        }
|    }
|
|    int get_degree(int n) {
|        int num_zeros = __builtin_clz(n-1);
|        return 32-num_zeros;
|    }
|
|    vector<int> multiply(vector<int> a, vector<int> b) {
|        int len = (int)a.size()+(int)b.size()-1;
|        int log1 = get_degree(len), n = (1 << log1);
|        a.resize(n, 0), b.resize(n, 0);
|        FFT(a, log1), FFT(b, log1);
|        for (int q = 0; q < n; q++) {
|            a[q] *= b[q], a[q] %= C;
|        }
|        IFFT(a, log1), a.resize(len);
|        return a;
|    }
|
|    void add_seg(vector<int>& a, vector<int>& b, vector<int>& c, int q, int len, int r) {
|        vector<int> a1(a.begin()+q-len, a.begin()+q);
```

```
        vector<int> b1(b.begin(), b.begin()+min(r, 2*len));
        vector<int> c1 = multiply(a1, b1);
        for (int q1 = 0; q1 < min(r, len); q1++) {
            c[q+q1] += c1[q1+len-1];
        }
    }

    vector<int> online_FFT(vector<int>& b, int c0, int n) {
        vector<int> c(2*n+1, 0); // if a == c
        c[0] = c0;
        for (int q = 1; q <= n; q++) {
            int deg = __builtin_ctz(q), len = (1 << deg);
            add_seg(c, b, c, q, len, q);
            add_seg(b, c, c, q, len, q-len);
            int rev_q = pow1(q, C-2); // for Bell's numbers
            c[q] = c[q] % C*rev_q % C;
        }
        c.resize(n+1);
        return c;
    }
```

C_k_n_any_modulo

```
    int phi(int n, const vector<int> primes) {
        int ans = n;
        for (int q : primes) {
            ans -= ans/q;
        }
        return ans;
    }

    vector<int> fact, rev_fact;

    void make_fact(int n, int C, vector<int> primes) {
        fact = {1};
        vector<int> numbers = {0};
        for (int q = 1; q <= n; q++) {
            int now = q;
            for (int q1 : primes) {
                while (now % q1 == 0) {
                    now /= q1;
                }
            }
            fact.push_back(fact.back()*now % C);
            numbers.push_back(now);
        }
        rev_fact = {pow1(fact.back(), phi(C, primes)-1, C)};
        for (int q = min(n, C-1); q > 0; q--) {
            rev_fact.push_back(rev_fact.back()*numbers[q] % C);
        }
        reverse(rev_fact.begin(), rev_fact.end());
    }

    int p_degree(int n, int C) {
        int ans = 0, deg = C;
        while (deg <= n) {
            ans += n/deg, deg *= C;
        }
        return ans;
    }

    int C_k_n(int k, int n, int C, vector<int> primes) {
        int ans = fact[n]*rev_fact[k] % C*rev_fact[n-k] % C;
        for (int q : primes) {
```

```
            int deg = p_degree(n, q)-p_degree(k, q)-p_degree(n-k, q);
            ans = ans*pow1(q, deg, C) % C;
        }
        return ans;
    }
```

N_th_root

```
    int find_g(int C) {
        vector<int> primes = factor(C-1);
        for (int q = 1;; q++) {
            bool flag = true;
            for (int q1 : primes) {
                if (pow1(q, (C-1)/q1, C) == 1) {
                    flag = false;
                    break;
                }
            }
            if (flag) {
                return q;
            }
        }
    }

    int g_degree(int x, int g, int C) {
        int sqrt1 = sqrt(C);
        unordered_map<int, int> a;
        a.reserve(sqrt1);
        int now = 1;
        for (int q = 0; q < sqrt1; q++) {
            a[now] = q;
            now = now*g % C;
        }
        int rev_sqrt = pow1(now, C-2, C);
        for (int q = 0; q <= C/sqrt1; q++) {
            if (a.find(x) != a.end()) {
                return q*sqrt1+a[x];
            }
            x = x*rev_sqrt % C;
        }
        return -1;
    }

    int phi(int n) {
        vector<int> fact = factor(n);
        int ans = n;
        for (int q : fact) {
            ans -= ans/q;
        }
        return ans;
    }

    p ax_by_c(int a, int b, int c) {
        int t = __gcd(a, b);
        if (c % t != 0) {
            return {-1, -1};
        }
        a /= t, b /= t, c /= t;
        int x = (c*pow1(a, phi(b)-1, b) % b+b) % b;
        int y = (c-a*x)/b;
        return {x, y};
    }

    int sqrt_b(int a, int b, int C) {
```

```
|        if (a == 0) {
|            return 0;
|        }
|        int g = find_g(C);
|        int deg_a = g_degree(a, g, C);
|        int x = ax_by_c(b, C-1, deg_a).first;
|        return (x == -1 ? -1 : pow1(g, x, C));
|    }
```

double

```
|    #define ld long double
|    const ld E = 1e-8;
|
|    ld angle(Pt x, Pt y) {
|        // angle between (1, 0) и (sin, cos)
|        return atan2(cross(x, y), dot(x, y));
|    }
|
|    struct Line {
|        ld a, b, c;
|
|        Line(ld a, ld b, ld c): a(a), b(b), c(c) {}
|        Line(Pt x, Pt y): a(x.y-y.y), b(y.x-x.x), c(cross(x, y)) {}
|
|        Pt dir() const {
|            return {b, -a};
|        }
|
|        Pt norm() const {
|            return {a, b};
|        }
|
|        Line per(Pt x) const {
|            return {x, x+norm()};
|        }
|
|        bool on(Pt x) const {
|            return abs(a*x.x+b*x.y+c) < E;
|        }
|    };
|
|    bool on_line(Pt x, Pt y, Pt z) {
|        return abs(cross(y-x, z-x)) < E;
|    }
|
|    bool is_parallel(Line a, Line b) {
|        return abs(cross(a.dir(), b.dir())) < E;
|    }
|
|    vector<Pt> inter(Line a, Line b) {
|        if (is_parallel(a, b)) {
|            return {};
|        }
|        ld det = cross(a.norm(), b.norm());
|        ld det_x = cross(Pt(-a.c, a.b), Pt(-b.c, b.b));
|        ld det_y = cross(Pt(a.a, -a.c), Pt(b.a, -b.c));
|        return {Pt(det_x/det, det_y/det)};
|    }
|
|    Pt projection(Line line, Pt x) {
|        return inter(line, line.per(x))[0];
|    }
|
```

```
|    struct Seg {
|        Pt x, y;
|
|        Seg() = default;
|        Seg(Pt x, Pt y): x(x), y(y) {}
|
|        explicit operator bool() const {
|            return x != y;
|        }
|
|        Line line() const {
|            return {x, y};
|        }
|
|        bool on(Pt point) const {
|            return on_line(x, y, point) && dot(point-x, point-y) < E;
|        }
|    };
|
|    vector<Pt> inter(Seg a, Seg b) {
|        vector<Pt> inters;
|        if (!a && b.on(a.x)) {
|            inters.push_back(a.x);
|        } else if (!b && a.on(b.x)) {
|            inters.push_back(b.x);
|        }
|        if (!a || !b) {
|            return inters;
|        }
|        inters = inter(a.line(), b.line());
|        if (!inters.empty() && a.on(inters[0]) && b.on(inters[0])) {
|            return inters;
|        }
|        return {};
|    }
|
|    struct Cir {
|        Pt x;
|        ld r = 0;
|
|        Cir() = default;
|        Cir(Pt x, ld r): x(x), r(r) {}
|
|        bool on(Pt point) const {
|            return abs(dist(point, x)-r) < E;
|        }
|    };
|
|    vector<Pt> inter(Cir cir, Line line) {
|        Pt proj = projection(line, cir.x);
|        ld h = dist(proj, cir.x);
|        if (h > cir.r-E) {
|            return h > cir.r+E ? vector<Pt>() : vector{proj};
|        }
|        ld len = sqrtl(cir.r*cir.r-h*h);
|        Pt dir = line.dir().norm(len);
|        vector<Pt> ans = {proj-dir, proj+dir};
|        sort(ans.begin(), ans.end());
|        return ans;
|    }
|
|    vector<Pt> inter(Cir cir, Seg seg) {
|        if (!seg) {
|            return cir.on(seg.x) ? vector{seg.x} : vector<Pt>();
```

```
        }
        vector<Pt> ans, inters = inter(cir, seg.line());
        for (Pt q : inters) {
            if (seg.on(q)) {
                ans.push_back(q);
            }
        }
        return ans;
    }

    Line rad_axis(Cir cir1, Cir cir2) {
        ld a1 = cir1.x.x, b1 = cir1.x.y, r1 = cir1.r;
        ld a2 = cir2.x.x, b2 = cir2.x.y, r2 = cir2.r;
        return {2*(a2-a1), 2*(b2-b1), a1*a1-a2*a2+b1*b1-b2*b2+r2*r2-r1*r1};
    }

    vector<Pt> inter(Cir cir1, Cir cir2) {
        if (cir1.x == cir2.x) {
            return {};
        }
        return inter(cir1, rad_axis(cir1, cir2));
    }
```

polygons

```
    #define ld long double

    ld angle(Pt x, Pt y) {
        // angle between (1, 0) и (sin, cos)
        return atan2(cross(x, y), dot(x, y));
    }

    bool on_line(Pt x, Pt y, Pt z) {
        return cross(y-x, z-x) == 0;
    }

    struct Seg {
        Pt x, y;

        Seg(Pt x, Pt y): x(x), y(y) {}

        auto operator<=>(const Seg& other) const = default;

        explicit operator bool() const {
            return x != y;
        }
    };

    bool on_seg(Seg q, Pt point) {
        return on_line(q.x, q.y, point) && dot(point-q.x, point-q.y) <= 0;
    }

    int S_tr_2(Pt x, Pt y, Pt z) {
        return cross(y-x, z-x);
    }

    struct Polygon {
        vector<Pt> a;

        explicit Polygon(vector<Pt> a_ = {}): a(std::move(a_)) {
            if (a.empty()) {
                return;
            }
            normalize();
```

```
|            if (a.size() > 1) {
|                a.push_back(a[0]);
|            }
|        }
|
|        void counterclockwise() {
|            if (S_2() < 0) {
|                reverse(a.begin(), a.end());
|            }
|        }
|
|        void normalize() {
|            int n = (int)a.size();
|            int ind = 1;
|            while (ind < n-1 && on_line(a[0], a[ind], a[ind+1])) {
|                ind++;
|            }
|            if (ind >= n-1) {
|                Pt min1 = *min_element(a.begin(), a.end());
|                Pt max1 = *max_element(a.begin(), a.end());
|                a = (min1 == max1 ? vector{min1} : vector{min1, max1});
|                return;
|            }
|            rotate(a.begin(), a.begin()+ind, a.end());
|            a.push_back(a[0]);
|            vector<Pt> will_a = {a[0]};
|            for (int q = 1; q < n; q++) {
|                if (!on_line(will_a.back(), a[q], a[q+1])) {
|                    will_a.push_back(a[q]);
|                }
|            }
|            a = will_a;
|        }
|
|        vector<Seg> get_edges() const {
|            int n = (int)a.size();
|            vector<Seg> edges;
|            for (int q = 1; q < n; q++) {
|                edges.emplace_back(a[q-1], a[q]);
|            }
|            return edges;
|        }
|
|        bool on_border(Pt point) const {
|            return ranges::any_of(get_edges(), [point](Seg q) {return on_seg(q, point);});
|        }
|
|        int belonging(Pt point) const { // 0 -> out, 1 -> in, 2 -> border
|            if (on_border(point)) {
|                return 2;
|            }
|            ld ang = 0;
|            for (Seg q : get_edges()) {
|                ang += angle(q.x-point, q.y-point);
|            }
|            // will be 0 or 2*pi
|            return abs(ang) > numbers::pi_v<ld>;
|        }
|
|        int S_2() const {
|            int ans = 0;
|            for (Seg q : get_edges()) {
|                ans += S_tr_2(a[0], q.x, q.y);
|            }
```

```
            return ans;
        }
    };

    bool need_pop_back(Pt x, vector<Pt>& ans, bool up) {
        int m = (int)ans.size(), sign = 2*up-1;
        return m >= 2 && cross(ans[m-1]-ans[m-2], x-ans[m-2])*sign >= 0;
    }

    void make_envelope(vector<Pt>& a, bool up) {
        int n = (int)a.size();
        vector<Pt> ans = {a[0]};
        for (int q = 1; q < n; q++) {
            while (need_pop_back(a[q], ans, up)) {
                ans.pop_back();
            }
            ans.push_back(a[q]);
        }
        a = ans;
    }

    Polygon convex_hull(vector<Pt> a) {
        int n = (int)a.size();
        sort(a.begin(), a.end());
        if (n == 0 || a[0] == a.back()) {
            return Polygon(n == 0 ? vector<Pt>{} : vector{a[0]});
        }
        vector<Pt> up = {a[0]}, down = {a[0]};
        for (int q = 1; q < n-1; q++) {
            int value = cross(a.back()-a[0], a[q]-a[0]);
            if (value > 0) {
                up.push_back(a[q]);
            } else if (value < 0) {
                down.push_back(a[q]);
            }
        }
        up.push_back(a.back());
        down.push_back(a.back());
        make_envelope(up, true);
        make_envelope(down, false);
        up.insert(up.end(), down.rbegin()+1, down.rend()-1);
        return Polygon(up);
    }
```

Binary_Gauss

```
    #include <tr2/dynamic_bitset>
    using Bitset = tr2::dynamic_bitset<>;

    struct Gauss {
        vector<Bitset> a;
        int n, m;

        explicit Gauss(auto& a_): n(a_.size()), m(a_[0].size()) {
            for (int q = 0; q < n; q++) {
                a.emplace_back(m);
                for (int q1 = 0; q1 < m; q1++) {
                    a[q][q1] = a_[q][q1];
                }
            }
        }

        void subtract(int q, int q1) {
            for (int q2 = 0; q2 < n; q2++) {
```

```
                if (q != q2 && a[q2][q1]) {
                    a[q2] ^= a[q];
                }
            }
        }

        void gauss() {
            for (int q = 0; q < n; q++) {
                int q1 = (int)a[q].find_first();
                if (q1 != m) {
                    subtract(q, q1);
                }
            }
        }

        auto annihilator() const {
            vector<int> leader(n);
            for (int q = 0; q < n; q++) {
                leader[q] = (int)a[q].find_first();
            }
            vector<Bitset> ans;
            for (int q1 = 0; q1 < m; q1++) {
                if (ranges::find(leader, q1) != leader.end()) {
                    continue;
                }
                ans.emplace_back(m+1);
                for (int q = 0; q < n; q++) {
                    ans.back()[leader[q]] = a[q][q1];
                }
                ans.back().resize(m);
                ans.back()[q1] = true;
            }
            return ans;
        }
    };
```

SLAE_solve_module

```
    const int C = 1000000007;

    int pow1(int x, int y) {
        if (y == 0) {
            return 1;
        }
        if (y % 2 == 0) {
            return pow1(x*x % C, y/2);
        }
        return pow1(x, y-1)*x % C;
    }

    struct Matrix {
        vector<vector<int>> a;
        int n, m;

        Matrix(vector<vector<int>> &a1) {
            n = a1.size(), m = a1[0].size(), a = a1;
        }

        void subtract(int q, int q1) {
            int del = pow1(a[q][q1], C-2);
            vector<int> norm;
            for (int q3 = 0; q3 < m; q3++) {
                if (a[q][q3] != 0) {
                    norm.push_back(q3);
```

```
|                 }
|             }
|             for (int q2 = q+1; q2 < n; q2++) {
|                 if (a[q2][q1] != 0) {
|                     int coef = a[q2][q1]*del % C;
|                     for (int q3 : norm) {
|                         a[q2][q3] -= a[q][q3]*coef % C;
|                         a[q2][q3] += (a[q2][q3] < 0)*C;
|                     }
|                 }
|             }
|         }
|
|         void gauss() {
|             int q = 0, q1 = 0;
|             while (q < n && q1 < m) {
|                 for (int q2 = q; q2 < n; q2++) {
|                     if (a[q2][q1] != 0) {
|                         swap(a[q], a[q2]);
|                         break;
|                     }
|                 }
|                 if (a[q][q1] == 0) {
|                     q1++;
|                     continue;
|                 }
|                 subtract(q++, q1++);
|             }
|         }
|
|         vector<int> solve() {
|             gauss();
|             int q = n;
|             bool flag = true;
|             while (flag && (q--) > 0) {
|                 for (int q1 : a[q]) {
|                     flag &= (abs(q1) == 0);
|                 }
|             }
|             if (q == -1) {
|                 return {C};
|             }
|             flag = true;
|             for (int q1 = 0; q1 < m-1; q1++) {
|                 flag &= (a[q][q1] == 0);
|             }
|             if (flag) {
|                 return {};
|             }
|             if (q < n-1 || n != m-1) {
|                 return {C};
|             }
|             vector<int> ans(n);
|             for (; q > -1; q--) {
|                 int sum1 = a[q][m-1];
|                 for (int q1 = q+1; q1 < m-1; q1++) {
|                     sum1 -= a[q][q1]*ans[q1] % C;
|                 }
|                 ans[q] = (sum1 % C+C) % C*pow1(a[q][q], C-2) % C;
|             }
|             return ans;
|         }
|     };
```

Berlekamp

```
|    const int C = 1791179179;
|
|    int pow1(int x, int y) {
|        if (y == 0) {
|            return 1;
|        }
|        if (y % 2 == 0) {
|            return pow1(x*x % C, y/2);
|        }
|        return pow1(x, y-1)*x % C;
|    }
|
|    vector<int> Berlekamp(vector<int> &rec) {
|        int n = rec.size(), q1 = 0;
|        while (q1 < n && rec[q1] == 0) {
|            q1++;
|        }
|        if (q1 == n) {
|            return {};
|        }
|        int t = rec[q1] % C, q2 = q1++;
|        vector<int> was, now = vector<int>(q1, 0);
|        for (; q1 < n; q1++) {
|            int d = -rec[q1] % C;
|            for (int q = 1; q <= now.size(); q++) {
|                d = (d+now[q-1]*rec[q1-q]) % C;
|            }
|            if (d == 0) {
|                continue;
|            }
|            vector<int> will = now;
|            while (will.size() < q1-q2+(int)was.size()) {
|                will.push_back(0);
|            }
|            int mul = d*pow1(t, C-2) % C;
|            will[q1-q2-1] = (will[q1-q2-1]+mul) % C;
|            for (int q = 0; q < was.size(); q++) {
|                will[q1-q2+q] = (will[q1-q2+q]-was[q]*mul) % C;
|            }
|            was = now, now = will, t = d, q2 = q1;
|        }
|        for (int& q : now) {
|            q = (q+C) % C, q -= (q > C/2)*C;
|        }
|        while (!now.empty() && now.back() == 0) {
|            now.pop_back();
|        }
|        return now;
|    }
|
|    int stupid(int n) {
|        //to do
|    }
|
|    int find_n(int n, bool flag = false) {
|        int k = 57;
|        vector<int> a = {0};
|        for (int q = 1; q < (flag ? n+1 : k); q++) {
|            a.push_back(stupid(q));
|        }
|        vector<int> rec = Berlekamp(a);
|        if (flag) {
```

```
            for (int q : rec) {
                cout << q << ' ';
            }
            cout << endl;
        }
        for (int q = k; q <= n; q++) {
            a.push_back(0);
            for (int q1 = 1; q1 <= rec.size(); q1++) {
                a.back() += a[q-q1]*rec[q1-1] % C;
            }
            a.back() = (a.back() % C+C) % C;
        }
        return a[n];
    }
```

Floor_sum

```
    int floor_sum(int a, int b, int k) { // [a/b]+...+[k*a/b]
        if (a >= b) {
            int t = a/b, r = a % b;
            return k*(k+1)/2*t+floor_sum(r, b, k);
        }
        if (a == 0) {
            return 0;
        }
        int m = k*a/b, total = k*m;
        int complement = floor_sum(b, a, m);
        int on_diag = k*gcd(a, b)/b;
        return total+on_diag-complement;
    }

    int floor_sum(int a, int b, int l, int r) {
        return floor_sum(a, b, max(0LL, r-1))-floor_sum(a, b, max(0LL, l-1));
    }

    int sum_mod(int a, int b, int l, int r) {
        int amount = r*(r-1)/2-l*(l-1)/2;
        return a*amount-b*floor_sum(a, b, l, r);
    }
```

Or_And_convolution

```
    template <bool Rev = false>
    vector<int> SOS(vector<int> a) {
        int n = (int)a.size();
        for (int q1 = 1; q1 < n; q1 <<= 1) {
            for (int q2 = q1; q2 < n; q2 += (q1 << 1)) {
                for (int q = q2; q < q2+q1; q++) {
                    if constexpr (!Rev) {
                        a[q] += a[q ^ q1];
                    } else {
                        a[q] -= a[q ^ q1];
                    }
                }
            }
        }
        for (int& q : a) {
            q = (q % C+C) % C;
        }
        return a;
    }

    vector<int> num_ones;
```

```
|
|    void make_num_ones(int n) {
|        num_ones.assign(1 << n, 0);
|        for (int q = 0; q < n; q++) {
|            num_ones[1 << q] = 1;
|        }
|        for (int q = 1; q < (1 << n); q++) {
|            int last_bit = q-(q & (q-1));
|            num_ones[q] = num_ones[q ^ last_bit]+num_ones[last_bit];
|        }
|    }
|
|    int SOS_value(int q, const vector<int> &a) {
|        int ans = (1-((num_ones[q] & 1) << 1))*a[0];
|        for (int q1 = q; q1 > 0; q1 = ((q1-1) & q)) {
|            ans += (1-((num_ones[q ^ q1] & 1) << 1))*a[q1];
|        }
|        return (ans % C+C) % C;
|    }
|
|    auto or_convolution_SOS(const vector<int> &a, const vector<int> &b) {
|        int n = (int)a.size();
|        vector<int> c(n);
|        for (int q = 0; q < n; q++) {
|            c[q] = a[q]*b[q] % C;
|        }
|        return c;
|    }
|
|    vector<int> or_convolution(const vector<int>& a, const vector<int>& b) {
|        return SOS<true>(or_convolution_SOS(SOS(a), SOS(b)));
|    }
|
|    vector<int> and_convolution(vector<int> a, vector<int> b) {
|        int n = (int)a.size(), ALL = n-1;
|        for (int q = 0; q < (n >> 1); q++) {
|            swap(a[q], a[q ^ ALL]);
|            swap(b[q], b[q ^ ALL]);
|        }
|        vector<int> c = or_convolution(a, b);
|        for (int q = 0; q < (n >> 1); q++) {
|            swap(c[q], c[q ^ ALL]);
|        }
|        return c;
|    }
```

Smiths_theory

```
|    vector<vector<int>> get_graph() {
|        int n, m;
|        cin >> n >> m;
|        vector<vector<int>> d(n);
|        for (int q = 0; q < m; q++) {
|            int x, y;
|            cin >> x >> y;
|            x--, y--;
|            d[x].push_back(y);
|        }
|        return d;
|    }
|
|    vector<int> get_smith(vector<vector<int>> &d) {
|        int n = (int)d.size();
|        vector<vector<int>> d1(n);
```

```
|        for (int q = 0; q < n; q++) {
|            for (int q1 : d[q]) {
|                d1[q1].push_back(q);
|            }
|        }
|        vector<int> smith(n, -1), all(n);
|        iota(all.begin(), all.end(), 0);
|        bool continue1 = true;
|        for (int nim = 0; continue1; nim++) {
|            continue1 = false;
|            vector<int> num(n, 0);
|            for (int q : all) {
|                for (int q1 : d[q]) {
|                    num[q] += (smith[q1] == -1);
|                }
|            }
|            vector<p> is;
|            vector<bool> is_move(n, false), is_now(n, false);
|            for (int q : all) {
|                is.emplace_back(num[q], q);
|                is_now[q] = true;
|            }
|            sort(is.rbegin(), is.rend());
|            vector<int> ind(n);
|            for (int q = 0; q < is.size(); q++) {
|                ind[is[q].second] = q;
|            }
|            all = {};
|            while (!is.empty() && is.back().first == 0) {
|                continue1 = true;
|                int vertex = is.back().second;
|                is.pop_back();
|                if (!is_now[vertex]) {
|                    continue;
|                }
|                is_now[vertex] = false, smith[vertex] = nim;
|                for (int q : d1[vertex]) {
|                    if (is_move[q] || smith[q] != -1) {
|                        continue;
|                    }
|                    if (is_now[q]) {
|                        all.push_back(q);
|                        is_now[q] = false;
|                    }
|                    is_move[q] = true;
|                    for (int q1 : d1[q]) {
|                        if (!is_now[q1]) {
|                            continue;
|                        }
|                        auto w = lower_bound(is.rbegin(), is.rend(), p(num[q1], -INF));
|                        auto w1 = is.rbegin()+(int)is.size()-ind[q1]-1;
|                        w1->first--, num[q1]--;
|                        swap(*w, *w1);
|                        swap(ind[w->second], ind[w1->second]);
|                    }
|                }
|            }
|        }
|        return smith;
|    }
|
|    int sum_games_result(vector<vector<int>> &d1, vector<vector<int>> &d2,
|                         vector<int> &smith1, vector<int> &smith2, int x, int y) {
|        if (smith1[x] == -1 && smith2[y] == -1) {
```

```
            return -1;
        }
        if (smith1[x] != -1 && smith2[y] != -1) {
            return (smith1[x] ^ smith2[y]) == 0;
        }
        bool was_swap = false;
        if (smith1[x] != -1) {
            swap(d1, d2);
            swap(smith1, smith2);
            was_swap = true;
            swap(x, y);
        }
        bool flag = false;
        for (int q : d1[x]) {
            flag |= (smith1[q] == smith2[y]);
        }
        if (was_swap) {
            swap(d1, d2);
            swap(smith1, smith2);
        }
        return flag-1;
    }
```

CHT

```
    #define ld long double
    const ld E = 1e-8;

    struct Line {
        int k, b;

        int value(int x) const {
            return k*x+b;
        }

        ld value(ld x) const {
            return k*x+b;
        }
    };

    ld inter(Line a, Line b) {
        return (ld)(b.b-a.b)/(a.k-b.k);
    }

    struct CHT { // max, different angles
        deque<pair<Line, ld>> a;

        void add_increasing(Line line) {
            while (a.size() > 1 && a.back().first.value(a.back().second)-E <
                                   line.value(a.back().second)) {
                a.pop_back();
            }
            if (a.empty()) {
                a.emplace_back(line, -INF);
            } else {
                a.emplace_back(line, inter(a.back().first, line));
            }
        }

        void add_decreasing(Line line) {
            while (a.size() > 1 && a[1].first.value(a[1].second)-E < line.value(a[1].second)) {
                a.pop_front();
            }
            if (!a.empty()) {
```

```
|                a[0].second = inter(a[0].first, line);
|            }
|            a.emplace_front(line, -INF);
|        }
|
|        int ans(int x) const {
|            auto lambda = [](pair<Line, ld> x, ld y) {return x.second < y;};
|            auto w = --lower_bound(a.begin(), a.end(), x, lambda);
|            return w->first.value(x);
|        }
|    };
```

Li_Chao

```
|    struct Line {
|        int k, b;
|
|        Line(): k(0), b(INF) {}
|        Line(int k, int b): k(k), b(b) {}
|
|        int value(int x) const {
|            return k*x+b;
|        }
|    };
|
|    struct Node {
|        Line line;
|        Node *l, *r;
|
|        explicit Node(Line line): line(line), l(nullptr), r(nullptr) {}
|
|        void make_sons() {
|            if (l == nullptr) {
|                l = new Node(Line());
|            }
|            if (r == nullptr) {
|                r = new Node(Line());
|            }
|        }
|    };
|
|    struct Li_Chao { // min
|        Node* root;
|        int MIN, MAX;
|
|        Li_Chao(int MIN, int MAX): MIN(MIN), MAX(MAX), root(new Node(Line())) {}
|
|        static void add(int l, int r, Node* tree, Line line) {
|            int m = (l+r)/2;
|            if (line.value(m) < tree->line.value(m)) {
|                swap(line, tree->line);
|            }
|            if (r-l == 1) {
|                return;
|            }
|            tree->make_sons();
|            if (line.k < tree->line.k) {
|                add(m, r, tree->r, line);
|            } else {
|                add(l, m, tree->l, line);
|            }
|        }
|
|        void add(Line line) const {
```

```
|            add(MIN, MAX, root, line);
|        }
|
|        static int ans(int l, int r, Node* tree, int x) {
|            int answer = tree->line.value(x);
|            if (r-l == 1) {
|                return answer;
|            }
|            tree->make_sons();
|            int m = (l+r)/2;
|            if (x < m) {
|                answer = min(answer, ans(l, m, tree->l, x));
|            } else {
|                answer = min(answer, ans(m, r, tree->r, x));
|            }
|            return answer;
|        }
|
|        int ans(int x) const {
|            return ans(MIN, MAX, root, x);
|        }
|    };
```

Annealing_simulation

```
|    mt19937 randint(179);
|    const int MN = (1LL << 20);
|
|    bool P(int x, int y, ld t) {
|        ld is = exp((y-x)/t);
|        return randint() % MN < is*MN;
|    }
|
|    int get_score(int x, vector<int>& a) {
|        int ans = 0;
|        for (int q : a) {
|            ans += min(3LL, num_bits[q ^ x]);
|        }
|        return ans;
|    }
|
|    int delta_score(int ind, int x, vector<int> &a) {
|        int score = 0, w = a[ind];
|        a[ind] = x;
|        score += get_score(a[ind], a);
|        a[ind] = w;
|        score -= get_score(a[ind], a);
|        return score*2;
|    }
|
|    auto annealing(int n) {
|        vector<int> a(1 << k);
|        iota(a.begin(), a.end(), 0);
|        shuffle(a.begin(), a.end(), randint);
|        a.resize(n);
|        int score = 0;
|        for (int q : a) {
|            score += get_score(q, a);
|        }
|        ld t = 1000, gamma = 0.999;
|        ld prev_t = t;
|        while (t > 0.001) {
|            if (prev_t > 1.1*t) {
|                cerr << t << endl;
```

```
            prev_t = t;
        }
        int ind = randint() % a.size();
        int x = randint() % (1 << k);
        int delta = delta_score(ind, x, a);
        if (delta >= 0 || P(score, score+delta, t)) {
            a[ind] = x, score += delta;
        }
        t *= gamma;
    }
    return pair{score, a};
}
```

something_useful

```
__builtin_ffsll(x); // index of the last 1 (0 if x == 0)
__builtin_clzll(x); // number of leading zeros (x != 0)
__builtin_popcountll(x); // number of 1

bitset<17> b;
b._Find_first(); // index of first 1 in bitset
b._Find_next(3); // index of the next 1 in bitset




#pragma GCC optimize("O3,unroll-loops") // removes short cycles and repeats code of them instead
// [for (int q = 0; q < 3; q++) {ans += q;}] converts to [ans += 1, ans += 2, ans += 3]
#pragma GCC target("avx2,popcnt,lzcnt")
// avx2/avx/sse/sse2/sse3/sse4 -> do multiple operations in parallel in cycles
// popcnt/lzcnt/abm/bmi/bmi2 -> do some bitwise operations in 1 processor action




set(CMAKE_CXX_FLAGS -fsplit-stack) // lets you create very huge arrays and vectors
set(CMAKE_CXX_FLAGS -fsanitize=address,undefined) // helps to find RE and UB
add_compile_definitions(-D_GLIBCXX_DEBUG -DLOCAL)
```

STL_useful

```
#include <tr2/dynamic_bitset>
typedef tr2::dynamic_bitset<> Bitset;
Bitset a(3);




#include <bits/extc++.h>
#define int long long
#define ld long double
#define p pair<int, int>
#define endl '\n'
const int INF = (int)1e9+1;

using namespace __gnu_pbds;
using namespace std;

typedef tree<int, null_type, less<>, rb_tree_tag, tree_order_statistics_node_update> ordered_set;
```

```
    struct my_hash {
        const int seed = chrono::steady_clock::now().time_since_epoch().count();

        static int hash(int x) {
            x += 0x9e3779b97f4a7c15;
            x = (x ^ (x >> 30)) * 0xbf58476d1ce4e5b9;
            x = (x ^ (x >> 27)) * 0x94d049bb133111eb;
            return x ^ (x >> 31);
        }

        int operator()(int x) const {
            return hash(x+seed);
        }
    };

    gp_hash_table<int, null_type, my_hash> s;
```

Aho_Corasick

```
    const int E = 26;
    vector<vector<int>> d = {vector<int>(E, -1)};
    vector<int> term = {0}, suflink, num_term;
    const char FIRST = 'a';

    int add_symbol(int vertex, char w) {
        w -= FIRST;
        if (d[vertex][w] == -1) {
            d[vertex][w] = (int)d.size();
            d.emplace_back(E, -1);
            term.push_back(0);
        }
        return d[vertex][w];
    }

    int add_string(const string& s) {
        int vertex = 0;
        for (char q : s) {
            vertex = add_symbol(vertex, q);
        }
        term[vertex]++;
        return vertex;
    }

    int build_suflink(int vertex, int q) {
        int q1 = suflink[vertex];
        while (q1 != -1 && d[q1][q] == -1) {
            q1 = suflink[q1];
        }
        return q1 == -1 ? 0 : d[q1][q];
    }

    void make_suflinks() {
        suflink.assign(d.size(), -1), num_term.assign(d.size(), 0);
        queue<int> a;
        a.push(0);
        num_term[0] = term[0];
        while (!a.empty()) {
            int vertex = a.front();
            a.pop();
            for (int q_ = 0; q_ < E; q_++) {
                int q = d[vertex][q_];
                if (q != -1) {
                    suflink[q] = build_suflink(vertex, q_);
                    num_term[q] = term[q]+num_term[suflink[q]];
```

```
                    a.push(q);
                }
            }
        }
    }
```

Hash

```
    static constexpr int mods[] = {
            1791179179, 1791791791,
            1057057179057179057
    };

    static constexpr int muls[] = {
            17957179, 179,
            179179057,
    };

    template <typename T>
    struct Hash {
        const int C, D;
        vector<int> degs, hashs;

        Hash(const T& s, int type = 0): C(mods[type]), D(muls[type]) {
            degs = {1}, hashs = {0};
            for (auto q : s) {
                degs.push_back(degs.back()*D % C);
                hashs.push_back((hashs.back()*D+q) % C);
            }
        }

        int hash(int l, int r) const {
            int ans = (hashs[r]-hashs[l]*degs[r-l]) % C;
            return ans+(ans < 0)*C;
        }
    };
```

Manacher

```
    vector<int> Manacher(vector<int> &a1) {
        int n = a1.size();
        vector<int> a = {a1[0]};
        for (int q = 1; q < n; q++) {
            a.push_back(INF);
            a.push_back(a1[q]);
        }
        n = a.size();
        vector<int> man = {1};
        int ind = 0;
        for (int q = 1; q < n; q++) {
            int k = q;
            if (ind+man[ind] > q) {
                k = min(man[ind]+ind, q+man[2*ind-q]);
            }
            while (k < n && 2*q-k > -1 && a[k] == a[2*q-k]) {
                k++;
            }
            man.push_back(k-q);
            if (k > man[ind]+ind) {
                ind = q;
            }
        }
        for (int q = 0; q < n; q++) {
```

```
            if (a[q] == INF) {
                man[q] -= man[q] % 2;
            } else {
                man[q] -= 1-man[q] % 2;
            }
        }
        return man;
    }
```

Prefix_function

```
    vector<int> pref_func(const string& s) {
        int n = (int)s.size();
        vector<int> pref(n, 0);
        for (int q = 1; q < n; q++) {
            pref[q] = pref[q-1];
            while (pref[q] > 0 && s[pref[q]] != s[q]) {
                pref[q] = pref[pref[q]-1];
            }
            pref[q] += (s[pref[q]] == s[q]);
        }
        return pref;
    }
```

Suffix_array

```
    vector<int> suf_mas(string &s) {
        s += '#';
        int n = (int)s.size();
        vector<int> suf(n);
        iota(suf.begin(), suf.end(), 0);
        sort(suf.begin(), suf.end(), [&s](int x, int y) {return s[x] < s[y];});
        vector<int> cls(n, 0);
        for (int q = 1; q < n; q++) {
            bool more = (s[suf[q-1]] < s[suf[q]]);
            cls[suf[q]] = cls[suf[q-1]]+more;
        }
        int deg = 1;
        while (cls[suf.back()] < n-1) {
            int num_cls = cls[suf.back()];
            vector<int> nums(num_cls+1, 0);
            for (int q : suf) {
                int ind1 = q-deg+(q < deg)*n;
                nums[cls[ind1]]++;
            }
            vector<int> ind(num_cls+1, 0);
            for (int q = 1; q <= num_cls; q++) {
                ind[q] = ind[q-1]+nums[q-1];
            }
            vector<int> will_suf(n);
            for (int q : suf) {
                int ind1 = q-deg+(q < deg)*n;
                will_suf[ind[cls[ind1]]++] = ind1;
            }
            vector<int> will_cls(n, 0);
            for (int q = 1; q < n; q++) {
                bool start_dif = (cls[will_suf[q-1]] != cls[will_suf[q]]);
                int ind_prev = will_suf[q-1]+deg;
                ind_prev -= (ind_prev >= n)*n;
                int ind_my = will_suf[q]+deg;
                ind_my -= (ind_my >= n)*n;
                bool end_dif = (cls[ind_prev] != cls[ind_my]);
                bool change = (start_dif || end_dif);
```

```
|                 will_cls[will_suf[q]] = will_cls[will_suf[q-1]]+change;
|             }
|             suf = will_suf, cls = will_cls, deg *= 2;
|         }
|         return suf;
|     }
|
|     vector<int> LCP(const string& s, const vector<int>& sufmas) {
|         int n = (int)s.size()-1; // '#' in the end
|         vector<int> pos(n+1), lcp(n+1, 0);
|         for (int q = 0; q <= n; q++) {
|             pos[sufmas[q]] = q;
|         }
|         for (int q = 0; q <= n; q++) {
|             int q1 = pos[q];
|             if (q1 == n) {
|                 continue;
|             }
|             int next_q = sufmas[q1+1];
|             lcp[q1] = (q == 0 ? 0 : max(0LL, lcp[pos[q-1]] - 1));
|             while (s[q+lcp[q1]] == s[next_q+lcp[q1]]) {
|                 lcp[q1]++;
|             }
|         }
|         lcp.pop_back();
|         return lcp;
|     }
```

Suffix_automaton

```
|     struct Node {
|         int suf = -1, len = 0;
|         int term = 0, num = -1;
|     };
|
|     struct Automaton {
|         const int E = 10, FIRST = '0';
|         vector<vector<int>> d;
|         vector<Node> a;
|         int end;
|
|         void build_num(int vertex) {
|             a[vertex].num = (a[vertex].term > 0);
|             for (int q : d[vertex]) {
|                 if (q != -1 && a[q].num == -1) {
|                     build_num(q);
|                 }
|                 if (q != -1) {
|                     a[vertex].num += a[q].num;
|                 }
|             }
|         }
|
|         void build_other() {
|             while (end != 0) {
|                 a[end].term = a[end].len-a[a[end].suf].len;
|                 end = a[end].suf;
|             }
|             a[0].term = 1;
|             build_num(0);
|         }
|
|         Automaton() {
|             end = add_node();
```

```
|        }
|
|        explicit Automaton(const string& s): Automaton() {
|            for (char q : s) {
|                add(q);
|            }
|            build_other();
|        }
|
|        int add_node() {
|            d.emplace_back(E, -1);
|            a.emplace_back();
|            return (int)a.size()-1;
|        }
|
|        int clone_node(int q) {
|            d.push_back(d[q]);
|            a.push_back(a[q]);
|            return (int)a.size()-1;
|        }
|
|        void add(char w) {
|            w -= FIRST;
|            int vertex = end;
|            end = add_node();
|            a[end].len = a[vertex].len+1;
|            while (vertex != -1 && d[vertex][w] == -1) {
|                d[vertex][w] = end;
|                vertex = a[vertex].suf;
|            }
|            int Q = (vertex == -1 ? 0 : d[vertex][w]);
|            if (vertex == -1 || a[Q].len == a[vertex].len+1) {
|                a[end].suf = Q;
|                return;
|            }
|            int new_Q = clone_node(Q);
|            a[new_Q].len = a[vertex].len+1;
|            while (vertex != -1 && d[vertex][w] == Q) {
|                d[vertex][w] = new_Q;
|                vertex = a[vertex].suf;
|            }
|            a[Q].suf = a[end].suf = new_Q;
|        }
|
|        int go(const string& t) const {
|            int vertex = 0;
|            for (char q : t) {
|                q -= FIRST;
|                vertex = d[vertex][q];
|                if (vertex == -1) {
|                    return -1;
|                }
|            }
|            return vertex;
|        }
|    };
```

Z_function

```
vector<int> z_func(const string& s) {
    int n = (int)s.size();
    vector<int> z(n, 0);
    int ind_max = 0, right_max = 1;
    for (int q = 1; q < n; q++) {
        if (q+z[q-ind_max] < right_max) {
            z[q] = z[q-ind_max];
            continue;
        }
        while (right_max < n && s[right_max-q] == s[right_max]) {
            right_max++;
        }
        z[q] = right_max-q;
        ind_max = q, right_max += (z[q] == 0);
    }
    return z;
}
```

Persistent_DO

```
struct Node {
    int x;
    Node *l, *r;

    Node(int x1 = 0): x(x1), l(nullptr), r(nullptr) {}
};

void update(Node* tree) {
    tree->x = (tree->l != nullptr ? tree->l->x : 0)+(tree->r != nullptr ? tree->r->x : 0);
}

Node* build(int l, int r) {
    Node* now = new Node();
    if (r-l == 1) {
        return now;
    }
    int m = (l+r)/2;
    now->l = build(l, m), now->r = build(m, r);
    update(now);
    return now;
}

Node* change(Node* tree, int l, int r, int q, int x) {
    if (r-l == 1) {
        return new Node(x);
    }
    Node* now = new Node();
    int m = (l+r)/2;
    if (q < m) {
        now->l = change(tree->l, l, m, q, x), now->r = tree->r;
    } else {
        now->l = tree->l, now->r = change(tree->r, m, r, q, x);
    }
    update(now);
    return now;
}

int ans(Node* tree, int l, int r, int l1, int r1) {
    if (l1 >= r || l >= r1) {
        return 0;
    }
    if (l1 <= l && r <= r1) {
```

```
|            return tree->x;
|        }
|        int m = (l+r)/2;
|        return ans(tree->l, l, m, l1, r1)+ans(tree->r, m, r, l1, r1);
|    }
```

DO_bottom_up

```
|    struct DO {
|        vector<int> do_arr, mins;
|        vector<bool> degs;
|        int len;
|
|        DO(vector<int> a) {
|            len = 1;
|            while (len < a.size()) {
|                len <<= 1;
|            }
|            do_arr.assign(len << 1, 0), mins.assign(len << 1, INF);
|            degs.assign(len << 1, false), degs[len] = true;
|            for (int q = len; q < a.size()+len; q++) {
|                do_arr[q] = mins[q] = a[q-len];
|            }
|            for (int q = len-1; q > 0; q--) {
|                do_arr[q] = do_arr[q << 1]+do_arr[(q << 1)+1];
|                mins[q] = min(mins[q << 1], mins[(q << 1)+1]);
|                degs[q] = degs[q << 1];
|            }
|        }
|
|        void change(int q, int x) {
|            do_arr[q+len] = x, mins[q+len] = x;
|            q = ((q+len) >> 1);
|            while (q > 0) {
|                do_arr[q] = do_arr[q << 1]+do_arr[(q << 1)+1];
|                mins[q] = min(mins[q << 1], mins[(q << 1)+1]);
|                q >>= 1;
|            }
|        }
|
|        int ans(int l, int r) {
|            if (l >= r) {
|                return 0;
|            }
|            l += len, r += len-1;
|            int sum1 = 0;
|            while (l < r) {
|                sum1 += (l & 1)*do_arr[l];
|                l = ((l+1) >> 1);
|                sum1 += ((r & 1) ^ 1)*do_arr[r];
|                r = ((r-1) >> 1);
|            }
|            return sum1+(l == r)*do_arr[l];
|        }
|
|        int left_less(int q, int x) {
|            q += len;
|            while (!degs[q] && mins[q] >= x) {
|                q = ((q-1) >> 1);
|            }
|            if (mins[q] >= x) {
|                return -1;
|            }
|            while (q < len) {
```

```
|                q = (q << 1)+(mins[(q << 1)+1] < x);
|            }
|            return q-len;
|        }
|    };
```

Implicit_DO

```
|    struct Node {
|        int left, right;
|        Node *l, *r;
|        int size;
|
|        Node(int left1, int right1):
|            left(left1), right(right1), l(nullptr), r(nullptr), size(0) {}
|    };
|
|    void make_sons(Node* tree) {
|        int m = ((tree->left+tree->right) >> 1);
|        if (tree->l == nullptr) {
|            tree->l = new Node(tree->left, m);
|        }
|        if (tree->r == nullptr) {
|            tree->r = new Node(m, tree->right);
|        }
|    }
|
|    void add(Node* tree, int x) {
|        int l = tree->left, r = tree->right;
|        tree->size++;
|        if (r-l == 1) {
|            return;
|        }
|        make_sons(tree);
|        int m = ((l+r) >> 1);
|        if (x < m) {
|            add(tree->l, x);
|        } else {
|            add(tree->r, x);
|        }
|    }
|
|    void del(Node* tree, int x) {
|        int l = tree->left, r = tree->right;
|        tree->size--;
|        if (r-l == 1) {
|            return;
|        }
|        make_sons(tree);
|        int m = ((l+r) >> 1);
|        if (x < m) {
|            del(tree->l, x);
|        } else {
|            del(tree->r, x);
|        }
|    }
|
|    int num_x(Node* tree, int x) {
|        int l = tree->left, r = tree->right;
|        if (r-l == 1) {
|            return tree->size*(x == l);
|        }
|        make_sons(tree);
|        int m = ((l+r) >> 1);
```

```
        if (x < m) {
            return num_x(tree->l, x);
        }
        return num_x(tree->r, x)+tree->l->size;
    }
```

DD

```
    mt19937 randint(179);

    struct Node {
        int x, y;
        Node *l, *r;
        int size;

        Node(int x1): x(x1), y(randint()), l(nullptr), r(nullptr), size(1) {}
    };

    void update(Node* tree) {
        if (tree == nullptr) {
            return;
        }
        tree->size = 1;
        if (tree->l != nullptr) {
            tree->size += tree->l->size;
        }
        if (tree->r != nullptr) {
            tree->size += tree->r->size;
        }
    }

    Node* merge(Node* tree1, Node* tree2) {
        if (tree1 == nullptr) {
            return tree2;
        }
        if (tree2 == nullptr) {
            return tree1;
        }
        if (tree1->y < tree2->y) {
            tree1->r = merge(tree1->r, tree2);
            update(tree1);
            return tree1;
        }
        tree2->l = merge(tree1, tree2->l);
        update(tree2);
        return tree2;
    }

    pair<Node*, Node*> split(Node* tree, int x) {
        if (tree == nullptr) {
            return {nullptr, nullptr};
        }
        if (tree->x <= x) {
            pair<Node*, Node*> trees = split(tree->r, x);
            tree->r = trees.first;
            update(tree);
            return {tree, trees.second};
        }
        pair<Node*, Node*> trees = split(tree->l, x);
        tree->l = trees.second;
        update(tree);
        return {trees.first, tree};
    }
```

```
|    pair<Node*, Node*> split_num(Node* tree, int k) {
|        if (tree == nullptr) {
|            return {nullptr, nullptr};
|        }
|        int t = (tree->l != nullptr ? tree->l->size : 0);
|        if (t < k) {
|            pair<Node*, Node*> trees = split_num(tree->r, k-t-1);
|            tree->r = trees.first;
|            update(tree);
|            return {tree, trees.second};
|        }
|        pair<Node*, Node*> trees = split_num(tree->l, k);
|        tree->l = trees.second;
|        update(tree);
|        return {trees.first, tree};
|    }
|
|    Node* add(Node* tree, int x) {
|        pair<Node*, Node*> trees = split(tree, x);
|        Node* now = new Node(x);
|        return merge(merge(trees.first, now), trees.second);
|    }
|
|    Node* del(Node* tree, int x) {
|        pair<Node*, Node*> trees = split(tree, x);
|        int k = (trees.first != nullptr ? trees.first->size-1 : 0);
|        pair<Node*, Node*> trees1 = split_num(trees.first, k);
|        return merge(trees1.first, trees.second);
|    }
|
|    int num_x(Node* tree, int x) {
|        int ans = 0;
|        while (tree != nullptr) {
|            if (tree->x <= x) {
|                ans += (tree->l != nullptr ? tree->l->size : 0)+1;
|                tree = tree->r;
|            } else {
|                tree = tree->l;
|            }
|        }
|        return ans;
|    }
```

Implicit_DD

```
|    mt19937 randint(179);
|
|    struct Node {
|        int x, y;
|        Node *l, *r, *parent;
|        int size, sum;
|
|        Node(int x1): x(x1), y(randint()), l(nullptr), r(nullptr), parent(nullptr), size(1), sum(x) {}
|    };
|
|    void update(Node* tree) {
|        if (tree == nullptr) {
|            return;
|        }
|        tree->size = 1, tree->sum = tree->x;
|        if (tree->l != nullptr) {
|            tree->size += tree->l->size, tree->sum += tree->l->sum;
|        }
|        if (tree->r != nullptr) {
```

```
|            tree->size += tree->r->size, tree->sum += tree->r->sum;
|        }
|    }
|
|    void change_parent(Node* tree, Node* parent) {
|        if (tree == nullptr) {
|            return;
|        }
|        tree->parent = parent;
|    }
|
|    void change_left(Node* tree, Node* left) {
|        if (tree == nullptr) {
|            return;
|        }
|        tree->l = left;
|        change_parent(left, tree);
|        update(tree);
|    }
|
|    void change_right(Node* tree, Node* right) {
|        if (tree == nullptr) {
|            return;
|        }
|        tree->r = right;
|        change_parent(right, tree);
|        update(tree);
|    }
|
|    Node* merge(Node* tree1, Node* tree2) {
|        if (tree1 == nullptr) {
|            return tree2;
|        }
|        if (tree2 == nullptr) {
|            return tree1;
|        }
|        if (tree1->y < tree2->y) {
|            change_parent(tree1->r, nullptr);
|            change_right(tree1, merge(tree1->r, tree2));
|            return tree1;
|        }
|        change_parent(tree2->l, nullptr);
|        change_left(tree2, merge(tree1, tree2->l));
|        return tree2;
|    }
|
|    pair<Node*, Node*> split(Node* tree, int k) {
|        if (tree == nullptr) {
|            return {nullptr, nullptr};
|        }
|        int t = (tree->l != nullptr ? tree->l->size : 0);
|        if (k <= t) {
|            change_parent(tree->l, nullptr);
|            pair<Node*, Node*> trees = split(tree->l, k);
|            change_left(tree, trees.second);
|            return {trees.first, tree};
|        }
|        change_parent(tree->r, nullptr);
|        pair<Node*, Node*> trees = split(tree->r, k-t-1);
|        change_right(tree, trees.first);
|        return {tree, trees.second};
|    }
|
|    Node* add(Node* tree, int k, Node* vertex) {
```

```
|        pair<Node*, Node*> trees = split(tree, k);
|        return merge(merge(trees.first, vertex), trees.second);
|    }
|
|    Node* del(Node* tree, int k) {
|        pair<Node*, Node*> trees1 = split(tree, k);
|        pair<Node*, Node*> trees2 = split(trees1.second, 1);
|        return merge(trees1.first, trees2.second);
|    }
|
|    Node* root(Node* tree) {
|        if (tree == nullptr) {
|            return nullptr;
|        }
|        while (tree->parent != nullptr) {
|            tree = tree->parent;
|        }
|        return tree;
|    }
|
|    int find_pos(Node* tree) {
|        if (tree == nullptr) {
|            return 0;
|        }
|        int ans = (tree->l != nullptr ? tree->l->size : 0);
|        while (tree->parent != nullptr) {
|            if (tree->parent->l != tree) {
|                ans += (tree->parent->l != nullptr ? tree->parent->l->size : 0)+1;
|            }
|            tree = tree->parent;
|        }
|        return ans;
|    }
|
|    Node* find_element(Node* tree, int k) {
|        if (tree == nullptr) {
|            return nullptr;
|        }
|        int t = (tree->l != nullptr ? tree->l->size : 0);
|        if (k == t) {
|            return tree;
|        }
|        if (k < t) {
|            return find_element(tree->l, k);
|        }
|        return find_element(tree->r, k-t-1);
|    }
```

Persistent_DD

```
|    mt19937 randint(179);
|
|    struct Node {
|        int x, size;
|        Node *l, *r;
|
|        Node(int x1): x(x1), size(1), l(nullptr), r(nullptr) {}
|    };
|
|    void update(Node* tree) {
|        if (tree == nullptr) {
|            return;
|        }
|        tree->size = 1;
```

```
|        if (tree->l != nullptr) {
|            tree->size += tree->l->size;
|        }
|        if (tree->r != nullptr) {
|            tree->size += tree->r->size;
|        }
|    }
|
|    Node* copy(Node* tree) {
|        if (tree == nullptr) {
|            return nullptr;
|        }
|        Node* now = new Node(tree->x);
|        now->l = tree->l, now->r = tree->r;
|        update(now);
|        return now;
|    }
|
|    Node* merge(Node* tree1, Node* tree2) {
|        if (tree1 == nullptr) {
|            return tree2;
|        }
|        if (tree2 == nullptr) {
|            return tree1;
|        }
|        if (randint() % (tree1->size+tree2->size) < tree1->size) {
|            Node* now = copy(tree1);
|            now->r = merge(tree1->r, tree2);
|            update(now);
|            return now;
|        }
|        Node* now = copy(tree2);
|        now->l = merge(tree1, tree2->l);
|        update(now);
|        return now;
|    }
|
|    pair<Node*, Node*> split(Node* tree, int k) {
|        if (tree == nullptr) {
|            return {nullptr, nullptr};
|        }
|        int left = (tree->l == nullptr ? 0 : tree->l->size);
|        Node* now = copy(tree);
|        if (k <= left) {
|            pair<Node*, Node*> trees = split(tree->l, k);
|            now->l = nullptr;
|            update(now);
|            trees.second = merge(trees.second, now);
|            return trees;
|        }
|        pair<Node*, Node*> trees = split(tree->r, k-left-1);
|        now->r = nullptr;
|        update(now);
|        trees.first = merge(now, trees.first);
|        return trees;
|    }
|
|    pair<Node*, bool> change(Node* tree, int pos) {
|        auto trees1 = split(tree, pos+1);
|        auto trees2 = split(trees1.first, pos);
|        Node* will = copy(trees2.second);
|        bool flag = (will->x == 1);
|        will->x = 1-will->x;
|        return {merge(merge(trees2.first, will), trees1.second), flag};
```

```
|    }
```

Fenwick

```
|    struct Fen {
|        vector<int> fen;
|        int n;
|
|        Fen(int n1) {
|            n = n1+1;
|            fen.assign(n, 0);
|        }
|
|        void plus(int q, int x) {
|            for (++q; q < n; q += (q & -q)) {
|                fen[q] += x;
|            }
|        }
|
|        int sum(int q) {
|            int res = 0;
|            for (; q > 0; q -= (q & -q)) {
|                res += fen[q];
|            }
|            return res;
|        }
|
|        int sum(int l, int r) {
|            return sum(r)-sum(l);
|        }
|    };
```

ICPC_algorithms

| ./graph/flow/Dinitz.cpp
| ./graph/flow/Ford_Fulkerson.cpp
| ./graph/flow/Min_cost.cpp
| ./graph/shortest_paths/Ford_Bellman.cpp
| ./graph/tree/LCA/LCA_linear_memory.cpp
| ./graph/tree/Compressed_tree.cpp
| ./graph/tree/HLD.cpp
| ./graph/Bridges.cpp
| ./graph/CSS.cpp
| ./graph/Points_articulation.cpp
| ./graph/Eulerian_cycle.cpp
| ./graph/Kun.cpp
| ./math/delivers/Pollard.cpp
| ./math/FFT/FFT_divide.cpp
| ./math/FFT/FFT_modulo.cpp
| ./math/FFT/online_FFT.cpp
| ./math/functions/C_k_n_any_modulo.cpp
| ./math/functions/N_th_root.cpp
| ./math/geometry/double.cpp
| ./math/geometry/polygons.cpp
| ./math/matrix/Binary_Gauss.cpp
| ./math/matrix/SLAE_solve_module.cpp
| ./math/Berlekamp.cpp
| ./math/Floor_sum.cpp
| ./math/Or_And_convolution.cpp
| ./math/Smiths_theory.cpp
| ./other/dp/CHT.cpp
| ./other/dp/Li_Chao.cpp
| ./other/Annealing_simulation.cpp
| ./other/something_useful.cpp
| ./other/STL_useful.cpp
| ./string/Aho_Corasick.cpp
| ./string/Hash.cpp
| ./string/Manacher.cpp
| ./string/Prefix_function.cpp
| ./string/Suffix_array.cpp
| ./string/Suffix_automaton.cpp
| ./string/Z_function.cpp
| ./struct/Segment_Tree/persistent/pointers.cpp
| ./struct/Segment_Tree/DO_bottom_up.cpp
| ./struct/Segment_Tree/Implicit_DO.cpp
| ./struct/Treap/DD.cpp
| ./struct/Treap/Implicit_DD.cpp
| ./struct/Treap/Persistent_DD.cpp
| ./struct/Fenwick.cpp